REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
Jun 2008	Conference Paper Postprint	26 Apr 08 – 23 Jun 08

4. TITLE AND SUBTITLE

PERFORMANCE OPTIMIZATION FOR PATTERN RECOGNITION USING ASSOCIATIVE NEURAL MEMORY

5a. CONTRACT NUMBER
In-House

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61102F

6. AUTHOR(S)

Qing Wu, Prakash Mukre, Richard Linderman, Thomas Renz, Daniel Burns, Michael Moore, Qinru Qiu

5d. PROJECT NUMBER
231T

5e. TASK NUMBER
IN

5f. WORK UNIT NUMBER
HP

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
AFRL/RITB
525 Brooks Road
Rome NY 13441-4505

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFRL/RITB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TP-2009-7

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited. PA# 88-ABW-2008-0062

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This paper describes the performance optimization in software and hardware solutions for a cognitive computing model called Brain State in a Box (BSB). This BSB model is implemented using two different configuration of the proposed architecture. The first implementation is a software only approach using the Cell Broadband Engine. The other implementation is a hybrid configurable computing platform which uses Field Programmable Gate Array (FPGA) for implementing the computation. To compensate its efficiency, the BSB based associative neural memory is applies for symbol and character recognition.

15. SUBJECT TERMS
Brain State in a Box model, software implementation, FPGA implementation, associative neural memory

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Stanley Lis
U	U	U	UU	15	19b. TELEPHONE NUMBER (*Include area code*) N/A

Performance Optimization for Pattern Recognition using Associative Neural Memory

Qing Wu[1], Prakash Mukre[1], Richard Linderman[2], Tom Renz[2], Daniel Burns[2], Michael Moore[3], Qinru Qiu[1]

[1]Department of Electrical and Computer Engineering, Binghamton University, Binghamton, NY 13902

[2]Air Force Research Laboratory, Rome Site, 26 Electronic Parkway, Rome, NY 13441

[3]ITT Advanced Engineering & Sciences, 775 Daedalian Drive, Rome, NY 13441

1 Introduction

Modeling and simulation of human cognizance functions involve large scale mathematical models, which demand high performance computing platform. We need a novel computing architecture which meets the computational capacity and communication bandwidth of a large scale associative neural memory model.

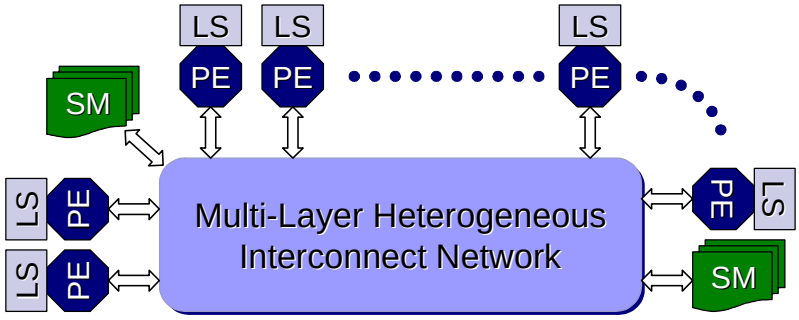


Figure 1. High performance cognitive computing system.

Figure 1 shows the targeting architecture of a high performance cognitive computing system. This system consists of multiple Processing Elements (PEs) interconnected with multi-layer heterogeneous interconnect network. All PEs have access to a local memory called Local Store (LS) and Shared Memory (SM) connected to the interconnect network.

In this paper we describe the performance optimization in software and hardware solutions for a cognitive computing model called *Brain State in a Box (BSB)*. This BSB model is implemented using two different configuration of the proposed architecture. The first implementation is a software only approach using the Cell Broadband Engine. The other implementation is a hybrid configurable computing platform which uses *Field Programmable Gate Array* (FPGA) for implementing the computation. To demonstrate its efficiency, the BSB based associative neural memory is applied for symbol and character recognition.

1.1 Brain State in a Box (BSB)

BSB model is a simple, auto-associative, nonlinear, energy-minimizing neural network [1][2]. A common application of the BSB model is to recognize a pattern from a given noisy version. BSB model can also be used as a pattern recognizer that employs a smooth nearness measure and generates smooth decision boundaries [3].

There are two main operations in a BSB model, *Training* and *Recall*. The mathematical model of a BSB ***recall operation*** can be represented in the following form

$$\mathbf{x}(t+1) = S(\alpha * \mathbf{A} * \mathbf{x}(t) + \lambda * \mathbf{x}(t) + \gamma * \mathbf{x}(0)) \quad (1)$$

Where:

- **x** is an *N* dimensional real vector

- **A** is an *NxN* connection matrix
- **A*****x** *(t)* is a matrix-vector multiplication operation
- α is a scalar constant feedback factor
- λ is an inhibition decay constant
- γ is a nonzero constant if there is a need to maintain the input stimulation
- *S ()* is the "squash" function defined as follows:

$$S(y) = \begin{cases} 1 & if \quad y \geq 1 \\ y & if \quad -1 < y < 1 \\ -1 & if \quad y \leq -1 \end{cases}$$

Note that in our implementation, we choose λ to be 1.0 and γ to be 0.0. But they can be easily changed to other values.
The ***training operation*** will use the following equation to determine the weight coefficients in **A**.

$$\Delta \mathbf{A} = lrate * (\mathbf{x} - \mathbf{A}\mathbf{x}) \otimes \mathbf{x} \quad (2)$$

$$\mathbf{A} = \mathbf{A} + \Delta \mathbf{A} \quad (3)$$

Where,

- **x** is the normalized input training pattern, a *N* dimensional real vector.
- *lrate* is the learning rate of the training operation.
- $\otimes$ is the operator for the outer product of two vectors.

Rest of the paper is organized as follows: Section 2 provides brief overview of the cell broadband engine architecture. Section 3 discusses the implementation details of BSB recall function on cell processor and experimental results. Section 4 presents the hardware implementation of BSB recall and training on a hybrid computing platform. Section 5 discusses the future work of scaling the BSB implementation in cell processor.

2 Cell Broadband Engine Architecture

Cell Broadband Engine Architecture [4][5][6] is a novel multi-core architecture designed for high performance computing. The architecture features nine microprocessors on single chip. A Power architecture compliant core called *Power Processing Element* (PPE) and 8 other attached processing cores called *Synergetic Processing Elements* (SPEs) are interconnected by high bandwidth *Element Interconnect Bus* (EIB). This heterogeneous architecture with high performance competing cores is designed for distributed computing.

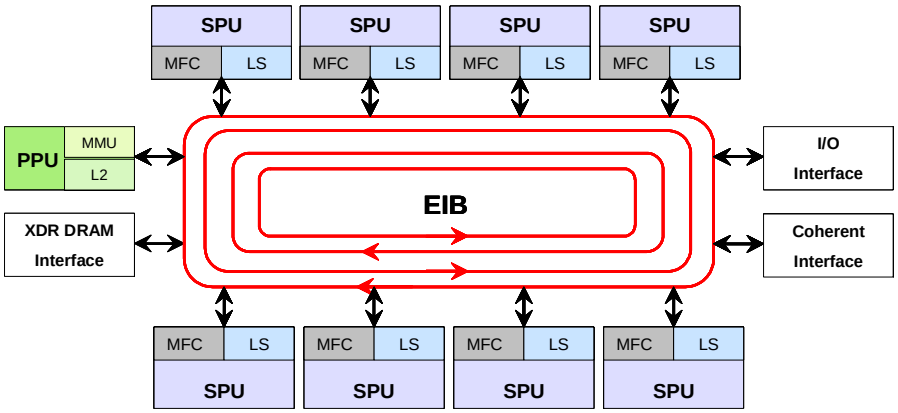


Figure 2. Cell Broadband Engine block diagram.

2.1 Power Processing Element (PPE)

The PPE is a 64bit, two-way SMT, PowerPC compatible processor. PPE acts as a main processor on which operating system runs and it has special functionalities to co-ordinate SPEs. Unlike other modern microprocessors PPE has a simplified in-order pipeline with no dynamic branch prediction. This reduces significant amount of hardware and thus enables PPE to run at higher frequency and yet consume less power. It has a simplified architecture which is more suitable for control functionality than high performance computing.

PPE supports two cache hierarchies with 32KB of instruction and data caches and a 512KB second level cache which preserves global coherence across the system.

2.2 Synergetic Processing Element (SPE)

The major computing power and performance of Cell processor comes from the 8 SPEs. The SPEs consist of new processor, designed to accelerate media and streaming applications. SPE mainly consists of *Synergetic Processing Unit* (SPU) and *Memory Flow Control* (MFC).

The SPU is a dual issue, in-order processor without any dynamic branch prediction. It has a large 128-entry register file. All registers are 128-bit Single Instruction Multiple Data (SIMD) registers. Most of the instructions provided by the SPU operate in a SIMD fashion on 128 bits of data. This 128bit data can be two 64-bit double floats or long integers, four 32-bit single floats or integers, eight 16-bit shorts, or 16 bytes. An instruction can source up to three 128-bit operands and produce one 128-bit result. So SPU register file supports 6 reads and 2 writes per cycle. Each SPU has a 256KB memory called *Local Store* (LS). All load and store instructions can only directly access the local store. All data and code used by the SPU must be located in its local store. SPU does not have any cache. However, load and stores have short latencies and are fully pipelined which enables the programmer to use local store as a software controlled cache.

In order to read and write data from the memory, SPE has to use explicit DMA instructions to initiate the transfer. Each SPE has a Memory Flow Controller (MFC) which handles the DMA operations. Once the SPU has initiated the DMA transfer, MFC carries it out in the background while SPU can continue to compute and access local store. SPU can poll or elect to receive an interrupt upon completion of DMA transfer. MFC supports up to 16 outstanding DMA commands. MFC uses the page tables on the PPE to perform address translations, which enables the SPE to access the entire virtual address space of the program through DMA. Each SPE's local store and other special communication channels are mapped into the memory map of the processor. Each SPE can initiate the DMA transfer between its local store and other SPEs local store.

DMA transfers have some alignment requirements: transfers must be 1, 2, 4, 8 and 16, or multiples of 16 bytes, up to 16KB. Transfers less than a quad-word must be between memory and local store that are aligned on multiples of transfer size; larger transfers require addresses that are quad-word-aligned.

2.3 Element Interconnect Bus (EIB)

EIB interconnects PPE, 8 SPEs, the off chip XDR DDR memory and the external I/O. It consists of an address bus and four 16B data buses. Two data bus run clockwise while other two are anticlockwise. EIB operates at the half the frequency of the processor and has a peak bandwidth of 205GB/s. The sustained data bandwidth will be lower than peak bandwidth, many factors influence the sustained bandwidth like location of the source and destination, interfering of the new transfer requests on the transfers in flight, and whether data transfer is between memory and LS or between LSs of different SPEs.

3 Implementing BSB Recall Operation on Cell Processor

This section describes the implementation and performance optimization of the recall operation on a single cell processor. One of the major challenges in implementing the recall operation in software is the high computation demand. For one 128-dimensional BSB model recall we need a matrix to vector multiplication which involves 16384 floating point multiplication and 16256 floating point additions. In addition, we also need 128 floating point multiplications for the feedback factor and 256 comparisons. A large scale associative neural model would

involve large number of BSB models (of the order of 100,000). We need a parallel distributed computational model which can perform many BSB recall operations in parallel.

Cell processor with high performance computing cores is designed for distributed computing. We can run one BSB recall on each SPE. Following sections describe the implementation details of a 128-dimensional BSB recall operation on one SPE.

3.1 Matrix-Vector Multiplication

Multiplication of a 128x128 matrix with 128x1 vector can be represented as follows

$$\begin{bmatrix} ax_0 \\ ax_1 \\ ax_2 \\ ax_3 \\ .. \\ .. \\ .. \\ ax_{127} \end{bmatrix} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & .. & .. & .. & a_{0,127} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & .. & .. & .. & a_{1,127} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & .. & .. & .. & a_{2,127} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & .. & .. & . & a_{3,127} \\ .. & .. & .. & .. & .. & .. & .. & .. \\ .. & .. & .. & .. & .. & .. & .. & .. \\ .. & .. & .. & .. & .. & .. & .. & . \\ a_{127,0} & a_{127,1} & a_{127,2} & a_{127,3} & .. & .. & .. & a_{127,127} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ .. \\ .. \\ .. \\ x_{127} \end{bmatrix} \quad (4)$$

Where,

$ax_0 = a_{0,0}*x_0 + a_{0,1}*x_1 + a_{0,2}*x_2 + \ldots + a_{0,127}*x_{127}$
$ax_1 = a_{1,0}*x_0 + a_{1,1}*x_1 + a_{1,2}*x_2 + \ldots + a_{1,127}*x_{127}$
$ax_2 = a_{2,0}*x_0 + a_{2,1}*x_1 + a_{2,2}*x_2 + \ldots + a_{2,127}*x_{127}$
..
..
$ax_{127} = a_{127,0}*x_0 + a_{127,1}*x_1 + a_{127,2}*x_2 + \ldots + a_{127,127}*x_{127}$

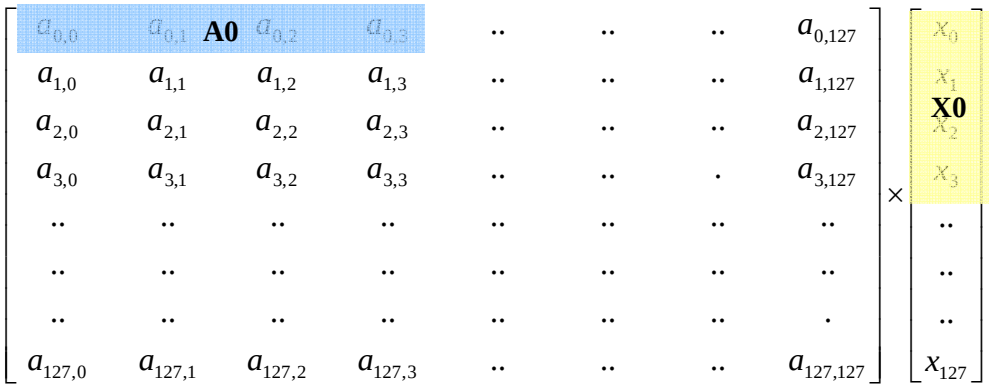
The scalar implementation of the above equations can be written in C as:

```
/* Example 1: Scalar matrix multiplication */

float ax[128], x[128], a[128*128];
int row,col;

for(row=0;row<128;row++){
   ax[row]=0;
   for(col=0;col<128;col++) {
      ax[row]+=x[col]*a[row*128+col];
   }
}
```

We can improve the performance by using the Single Instruction Multiple Data (SIMD) model of SPE. Most of the instructions in SPEs operate on 16 bytes of data and also the data fetching from local store is 16-byte aligned. To implement the scalar computation using SIMD instructions, complier has to keep track of the relative offsets between the scalar operands to get the correct results. This implementation ends up having additional rotation instructions added, which is an overhead. So, to get better performance and correct result it is wise to handle the data as vectors of 16 bytes (or 4 single-precision floating-point numbers) each.



The matrix multiplication using SIMD instructions can be implemented as below

```
/* Example 2: matrix multiplication using SIMD instructions*/

float ax[128] __attribute__((aligned (128)));
float x[128] __attribute__((aligned (128)));
float a[128*128] __attribute__((aligned (128)));
int row,col;
vector float  *x_v, *a_v;
vector float temp;
x_v = (vector float *)x;
a_v = (vector float *)a;


for(row=0;row<128/4;row++){
   temp=(vector float){0.0,0.0,0.0,0.0};
   for(col=0;col<128/4;col++) {
      temp=spu_madd(x_v[col],a_v[row*32+col],temp);
   }
   ax[row] = (spu_extract(temp,0)+spu_extract(temp,1)+
              spu_extract(temp,2)+spu_extract(temp,3));
}
```

spu_madd and *spu_extract* are intrinsics which make the underlying Instruction Set Architecture (ISA) and SPE hardware accessible from C programming language. *spu_madd* is the C representation for multiply and add instruction. *spu_extract* returns the vector member value specified by the offset. Compared to the scalar implementation, SIMD code reduces 16384 multiplication operations to 1024 vector multiplications. The above implementation still performs scalar addition to get the final result. We can further improve the performance by rearranging the matrix so that we can apply SIMD instructions on all the matrix-vector multiplication operations.

We divide the entire matrix into smaller 4x4 matrices. Elements of each of these 4x4 matrices are shuffled according to a specific pattern as shown below.

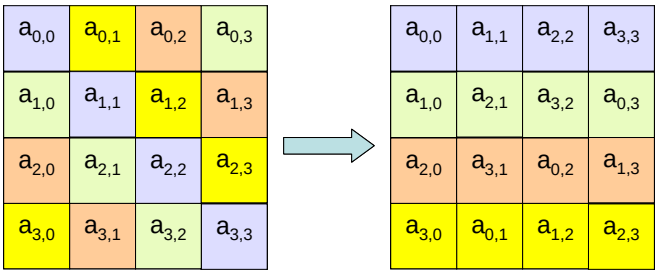


Figure 3. Matrix shuffling from original order to SIMD order.

The shuffled matrix is multiplied with the X vector as shown below. Note that each row of the shuffled matrix is a vector of 4 single precision floating point numbers. And (x_0, x_1, x_2, x_3) is the other vector in the SIMD operation.

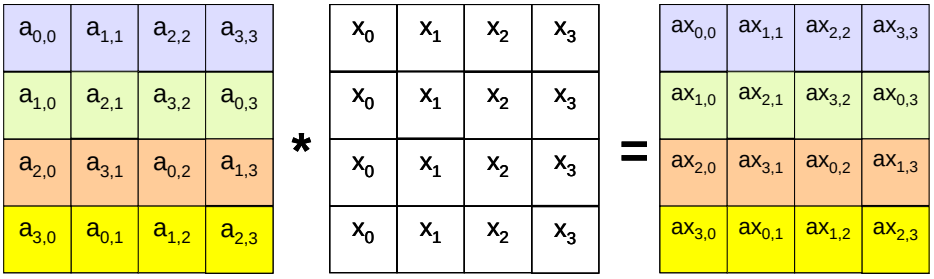


Figure 4. Shuffled matrix multiplication.

To obtain the final result we need to rotate the some of the elements to align them back to their original offset and add all the rows.

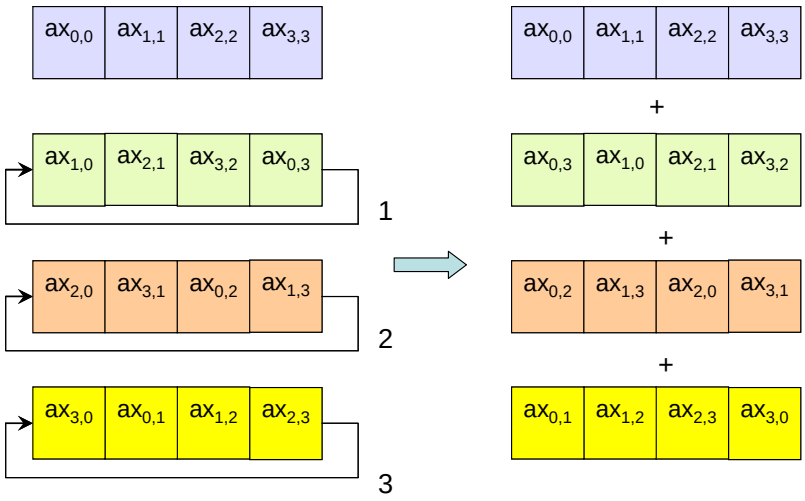


Figure 5. Product alignment and accumulation.

By shuffling the input matrix we have effectively replaced the 3*128 scalar additions from previous implementation with 3*32 rotations and also we reduced the overhead added by the compiler to perform scalar addition.

For every 4 rows of the 128x128 matrix, we will have 32 4x4 matrices. The above shuffle-multiply-rotation-accumulate operation is repeated for each one of them, and at the end we will be able to obtain the 4 elements of the resulting 128x1 vector. For 128 rows, we need to repeat the above procedure 32 times and obtain the complete

result. In our current program implemented on PS3, we assume that the 128x128 matrix has already been shuffled before being stored into the main memory. Therefore, the shuffle operations are not needed in the algorithm.

3.2 Other operations in BSB recall

From Equation (1) we know that **A*****x**(*t*) has to be multiplied with feed back constant α and the result is added with **x***(t)*. This multiplication and addition can be performed by using *spu_madd* intrinsic, which multiplies given two vectors and adds the result with the third vector. This requires 32 *spu_madd* intrinsics.

The final operation in recall is squash function. As given in Equation (1) this operation needs two comparisons to check whether the result of the previous computation is >1 or <-1. To perform this kind of compare and assign operations on vector data SPE provides special instructions.

spu_cmpgt is an intrinsic which performs element-wise comparison on given two vectors. If an element in first vector is greater than corresponding element in second vector then all the bits of corresponding element in result vector are set to '1' else '0'. This result can be used as a multiplexer select, which when '1' assigns input-1 to the output and when '0' assigns input0 to the output. An intrinsic called *spu_sel* is used to perform the selection. *spu_sel* takes two input vectors and select pattern. For each bit in the 128-bit vector pattern, the corresponding bits from either input vector-0 or input vector-1 is selected.

3.3 Performance Tuning

SPE has no dynamic branch prediction it always predicts the branch to be taken. Branches are detected late in the pipeline at a time where there are already multiple fall-through instructions in flight. Due to its architecture SPE has high branch misprediction penalty of 18 cycles. If we implement the multiplications as in Example 1 or 2, then at every loop entry there will be a branch misprediction. To reduce this branch miss penalty, we unrolled the entire inner loop including squash function. Apart from reducing branch misprediction, loop unrolling reduces dependencies and increases the dual-issue rates.

One of the important factors affecting the performance of SPE is the data transfer time. Due to limited local store size it is not possible to get all the data required for the computation at once. So SPE has to initiate DMA transfers whenever it requires additional data from the main memory, which takes additional time. However we can leverage the communication-computation concurrency provided by the Cell's asynchronous DMA model by performing computation while data for future computation is begin fetched. This is called *double buffering*. Figure 5 shows the computational flow with and without double buffering. In the regular communication model, the weight matrix required for the recall is fetched and 10 recall iterations are performed. Then the DMA request for the weight matrix of next recall is initiated. This induces holes in computational flow reducing the effective throughput. In the double buffered implementation, when the computation of the current recall starts, a DMA request for the weight matrix of next recall is initiated in parallel. The MFC can work in background to fetch the data from the memory. By the time when 10 iterations of current recall are completed, the data for the next recall will be available, so SPU can continue with its computational flow.

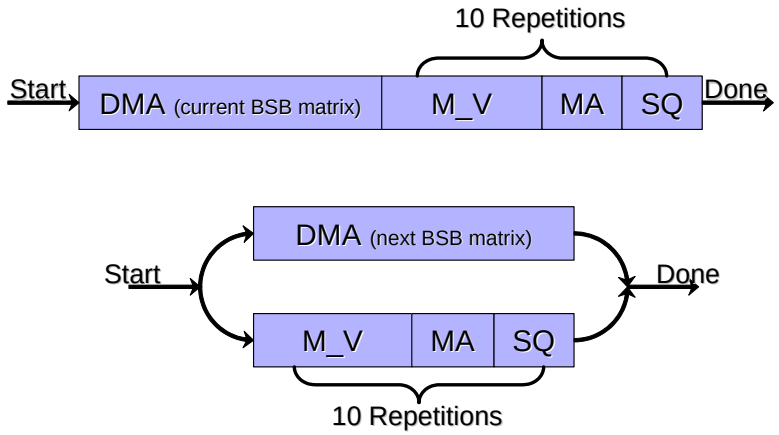


Figure 6. BSB recall computational flow without and with double buffering.

3.4 Experimental Results

A recall function for 128-neuron BSB model is implemented and tested on PlayStation®-3 (PS3). PS3 consists of one Cell Broadband Engine processor, 256MB XDR DRAM. Only six SPEs are available for the programmer, while the seventh is used by the Sony virtualization software and eighth is disabled. Fedora Core 6 (FC6) version of linux is installed on PS3, as well as the Cell *Software Development Kit* (SDK) 2.1.

We follow *function-offload model* of programming the cell processor. In this model SPEs are used as accelerators of computational intensive parts of the application. Main application runs on the PPE and it calls selected procedures to run on SPE. Multiple threads can be initiated to call different procedures on different SPEs. This way PPE can take advantage of asynchronous parallelism of SPEs. Both SPE and PPE source are compiled separately by different compilers. PPE passes the input parameters to the procedure running on SPE. Usually these parameters are either address of the data values for computation or synchronization signals.

In our implementation of BSB recall function, PPE sends the starting address of the weight matrices in main memory as parameter to the recall procedure running on SPE. After receiving the matrix address, SPE initiates the DMA transfer of weight matrix to its local store. We use double buffering as described in previous section, so that while the current recall is computed the next weight matrix will be fetched from the main memory through DMA. For testing purposes a random X vector is generated locally by the SPE.

PPE spawns 6 SPE threads once which implements 100,000 iterations of computation. For each iteration, the SPE performs the operation of loading a 128x128 weight matrix from memory and computing 10 BSB recall operations. To find out the computation/communication relationship of the algorithm, we have done different runs from 1 recall per iteration to 15 recalls per iteration. However, we keep the total number of recalls to be 1,000,000. For example, we run 100,000 iterations for 10 recalls per iteration, 200,000 iterations for 5 recalls per iteration. Figure 7 shows the performance in GFLOPS (Giga Floating-point Operations Per Second) per SPE obtained for the complete recall algorithm and Figure 8 shows the GFLOPS per SPE for only Matrix-vector multiplication part of the algorithm.

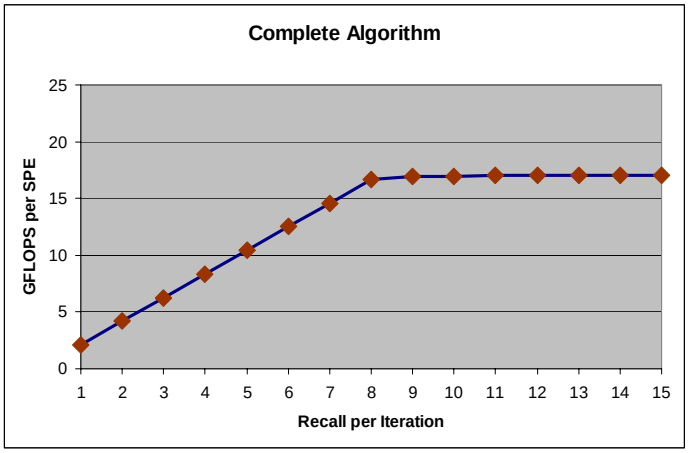


Figure 7. Performance for the whole recall operation.

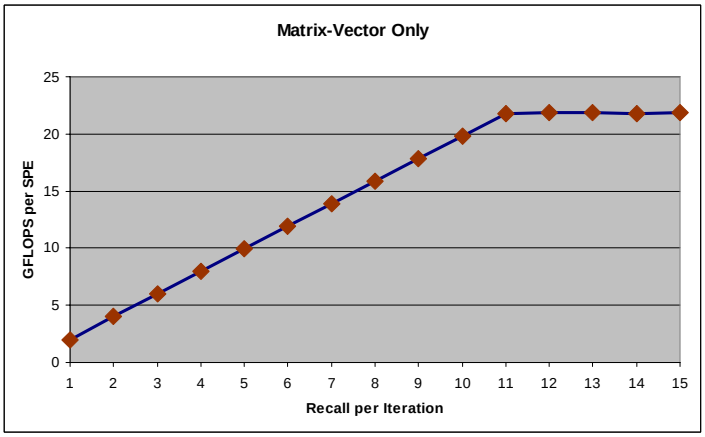


Figure 8. Performance for matrix-vector multiplication.

In both the graphs we can see that performance increases almost linearly to certain point and then saturates. In the linear region computation time is less than the communication time (even with double buffering), so SPE is idle till the weight matrix for the next recall is fetched. Since we use double buffering, the communication time is hidden by the computation time. This is the reason we get saturated performance when we computation time takes over communication time.

The complete source code of the implementation is attached at the end.

4 Hardware Implementation of BSB

In this section we describe hardware-based implementation for large scale BSB model. The architecture consists of a general purpose microprocessor and a FPGA. To address the high computational demand, we utilize the rich hardware resources like multiplier, adder, shift registers, etc, available in a FPGA to parallelize the computation. In addition we pipeline the design to increase the throughput.

4.1 BSB Recall

In this section we describe the hardware implementation of BSB recall as given in Equation (1). Figure 9 shows the data path of the recall hardware. The main computational elements in a BSB recall operation is matrix-vector multiplier. The matrix vector multiplier is implemented in three stage pipeline. Weight matrix is stored in 128 *Block RAMs (BRAMs)* called *W_BRAM*. The *read_addr* is used to index a particular row of the weight matrix. X vector is stored in two 128 stage shift registers, one of them is used for parallel multiplication and other is used for multiplying the *lambda*. The first stage of pipeline uses 128 multipliers in parallel to multiply the indexed row

of the matrix and X vector and result is store in P1 register. In second and third stages an adder tree is used to add all the 128 multiplication results. The result of the matrix multiplication is multiplied with *alpha* and stored in pipeline register P4. *lambda*X* is calculated using the 128 stage shift register. To meet the matrix-vector multiplier latency the result of *lambda*X* is delayed by 4 cycles using P5. Delayed value is added with P4 and squash function is applied.

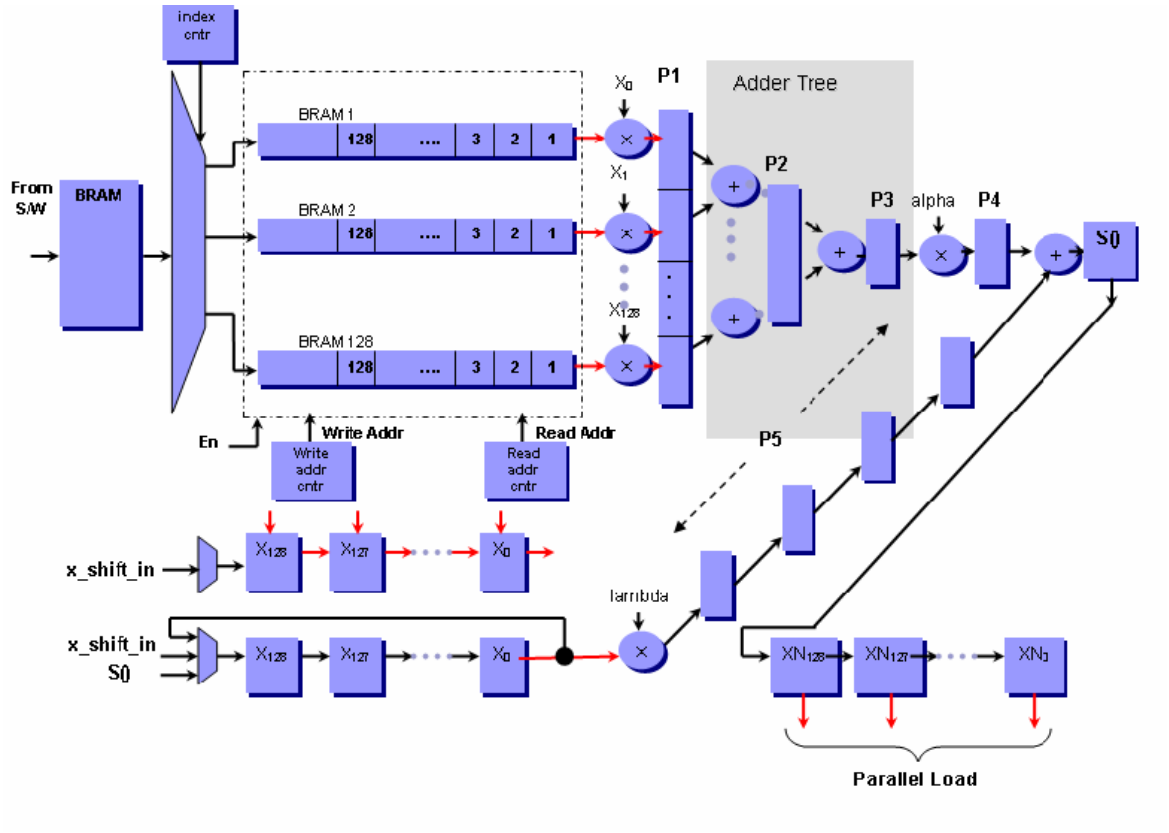


Figure 9. Data path of 128-neuron BSB recall operation.

4.2 BSB Training

Equation (2) and (3) give the mathematical model of a BSB training operation. Similar to recall, training operation involves matrix-vector multiplication and vector-vector outer product. Matrix-vector multiplier implementation is same as in recall operation. To optimize the resource utilization we have shared the 128 multipliers used in matrix-vector multiplication with vector-vector outer product calculation. A 2:1 multiplexer is used to selectively feed the input to the multipliers and a 1:2 decoder is used to route the result of multiplication either to an adder tree or a pipeline register P5.

Calculation can be divided into two phases. In the first phase the multiplexer selects the inputs from *W_BRAM* which contain the weight matrix initialized to '0' and decoder selects to route the result to adder tree. Register P3 contains the result of matrix-vector multiplication for the selected row. The result of ***lr* * (x – Ax)** is shifted into a 128 stage shift register. In the second phase, the output of this shift register is selected by the multiplexer to calculate the vector-vector outer product. The result of the product is added with the previous value of the weight matrix and weight matrix is updated.

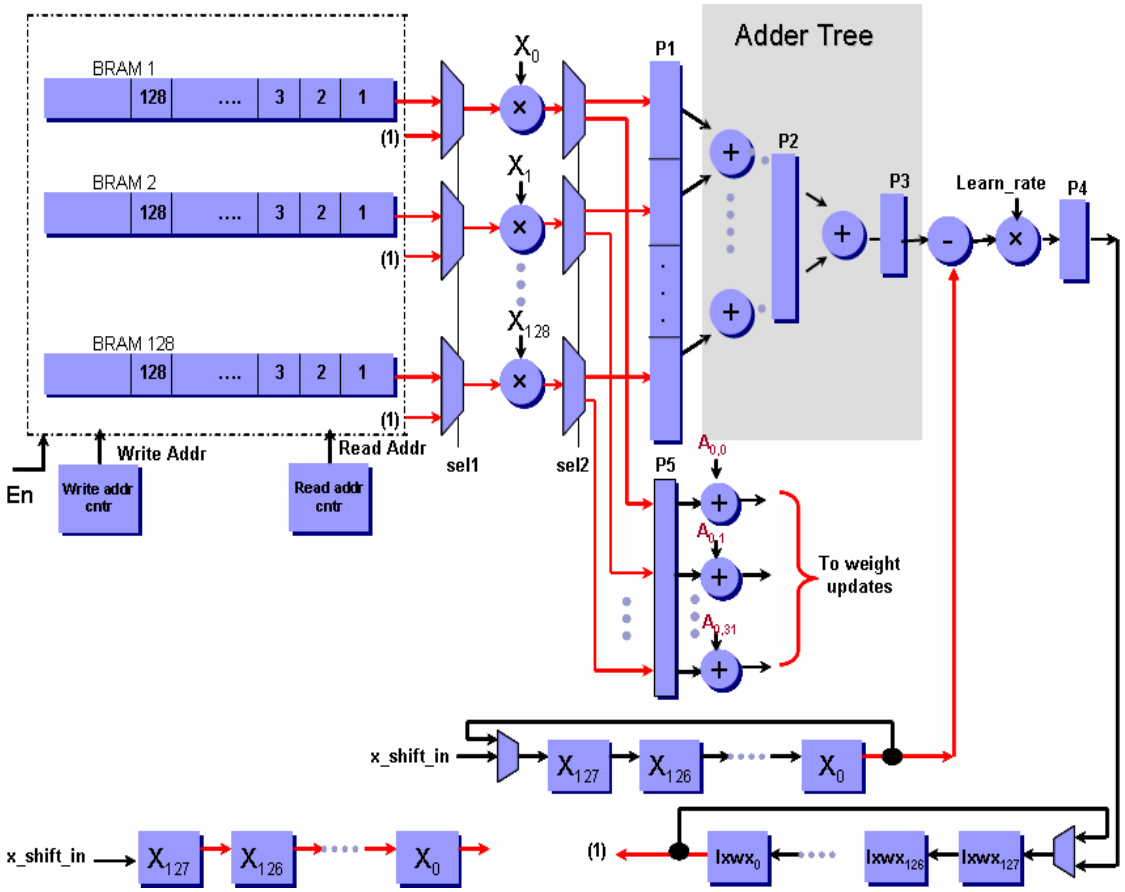


Figure 10. Data path of 128-neuron BSB training operation.

4.3 Experimental Results

BSB recall and training designs have been implemented on Annapolis WildStar II Pro PCIX card [7]. WildStar II card consists of one Xilinx Vertex II Pro XC2VP70 FPGA device [8][9]. XC2VP70 device contains 74,448 logic cells and 5,904Kb of BRAM. Table below summarizes the resource utilization of both training and recall synthesized with target frequency of 100MHz.

Resources	Recall Design	Training Design
Block Rams	129 (of 288)	129 (of 288)
18-bit Multipliers	130 (of 328)	130 (of 328)
LUTs	6127 (8%)	22716 (33%)

Table 1. Resource utilization for BSB training and recall operation.

The results of the hardware implementation are compared with a software model of the BSB recall operation which is run on a 2GHz Intel Xeon processor with 2GB RAM. The software model uses Intel Math Kernel library 9.0 to optimize the matrix operations. The hardware implementation is run on Annapolis card mentioned above at 90MHz. Table 2 shows the average performance comparison for 1,000,000 recalls.

Implementation	Time per Recall *(µs)	Equivalent FLOPS
Software	12.5	~2.6G
Hardware	1.69	~19.4G

Table 2. Hardware v/s Software performance comparison for BSB recall operation.

4.4 Architecture for large scale BSB recall operations

In the recall implementation described in 4.1, loading of weight matrix is done iteratively through command and response based communication. Since the interface BRAM can only accommodate 4 rows of the weight matrix, once the data from interface BRAM is copied into the *W_BRAM*, a request is sent to host software to send the next set of rows. This takes 32 iterations. If we have to implement recall for large number of BSB models then this kind of communication can add lot of overhead. We can not increase the number of interface BRAM dues to software addressing issues. So there is no room for overlapping communication with computation. The other possible solution is to have a large amount of low latency memory and load all the weight matrices required for computation into it first and then start the computation.

The WildStar-II pro card has enough SRAM on the board to accommodate weight matrices of 256 128-dimensional BSB models [8]. We store the weight matrix of all the models into 6 SRAM banks in parallel. 256 X vectors are stored in 32 BRAMs called *X_BRAM*. Loading of weight matrix into *W_BRAM* and X vector into shift registers is done in parallel. Then 10 recall iterations are performed and the resultant X vector is stored back at its original location in *X_BRAM*. Once all the recall operations are performed the results stored in *X_BRAM* is sent back to the host software.

The design is working correctly on our PC-FPGA platform, running 256 BSB models in a time-interleaved fashion.

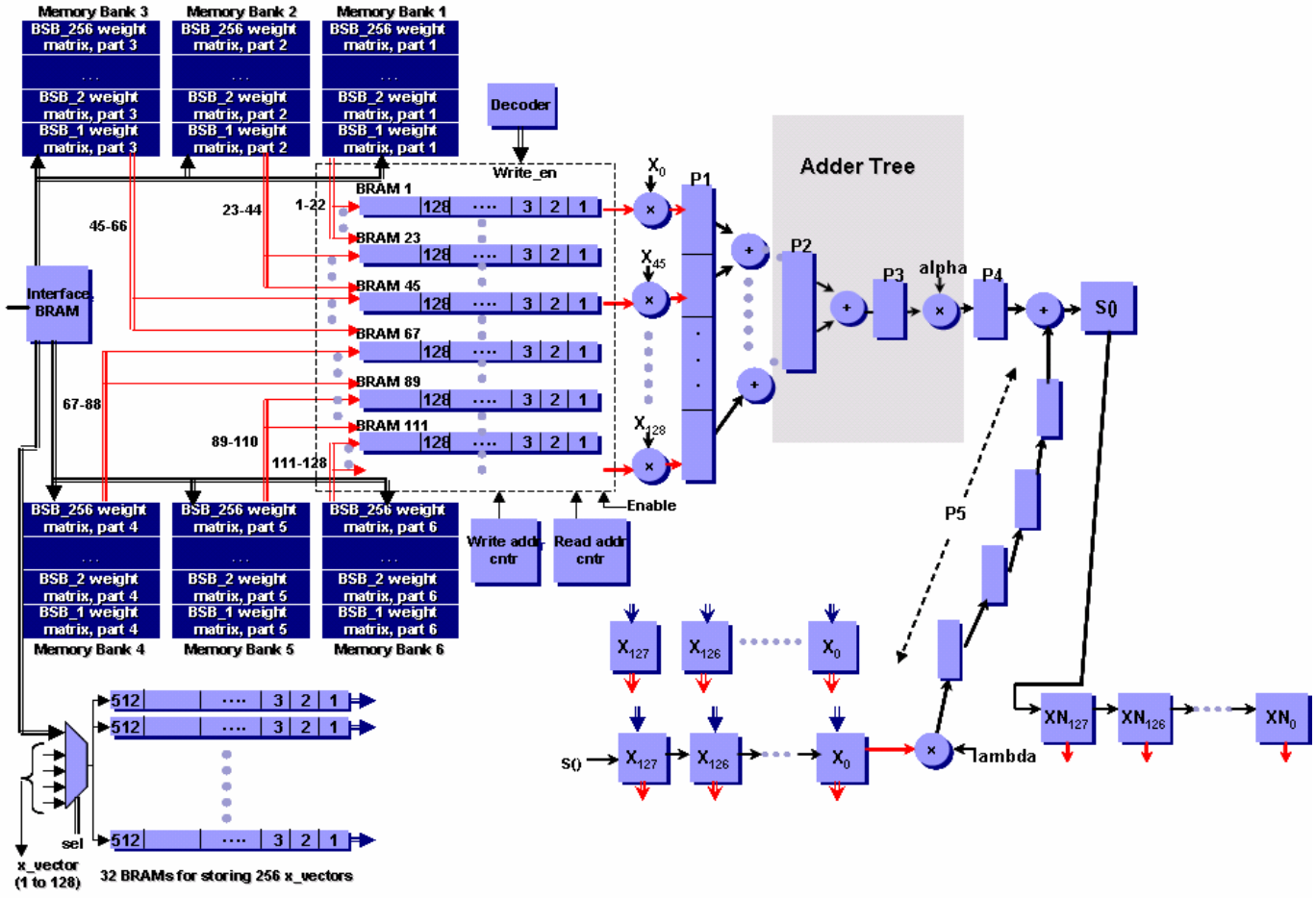


Figure 11. Hardware design for 256 128-dimensional BSB recall models.

5 Large-Scale BSB Models on PS3 Cluster

Due to limitation on the size of the local store of SPE, we can not linearly scale the size of the BSB model. For example to implement a 256-dimensional we need a matrix of 256x256 containing single precision floating point values. So the size of such a matrix would be 256*256*4 = 256K bytes. Apart from the matrix, we need storage for X vector, the result of multiplication, extra buffer of 256K bytes to implement double buffering, local variables and the code space. Apparently SPEs local store does not have so much space. To implement such a large model, either we have to share resources and serialize the computation or distribute it over multiple SPEs. We plan to distribute the recall operation over 4 SPEs, dividing the input weight matrix in 4 128x128 matrices. Each SPE can still use the matrix multiplication algorithm described before in this paper. In effect we can run one BSB model on one PS3. Many PS3 stations can be interconnected with a high speed network and the inter-BSB communication can be done with high level asynchronous communication protocol such as the publish/subscribe protocol or Message Passing Interface (MPI).

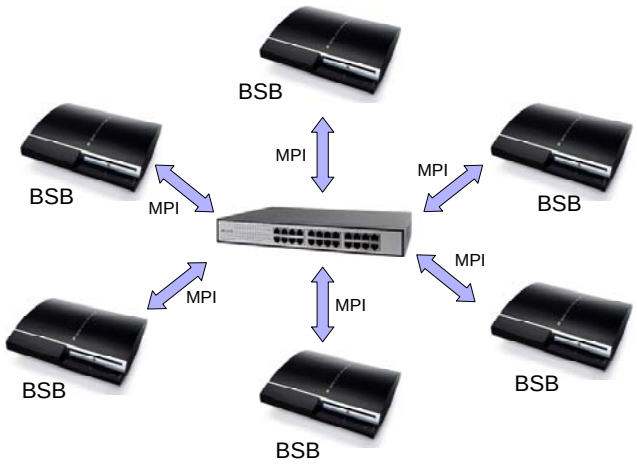


Figure 12. Large scale BSB model on a PS3 cluster.

A large scale BSB model finds application in many fields, one among them is pattern recognition. A BSB model can be used to retrieve text from noisy images, by training the model with all the alphabets of different font styles. Any alphabet, in an image, of a particular size can be treated as a fixed size matrix with its elements assuming values '+1' or '-1', where '+1' represents a black region in the alphabet and '-1' represents white region. We can use this information to train a BSB model for each letter in the alphabet. When a noisy image of an letter is given to recall, only the BSB model which was trained for this alphabet can converge in smaller number of iterations, thus identifying the correct alphabet. We can use this technique to extract the as many words as we can from a noisy image and then apply other cognitive algorithms, like confabulation to predict the missing text in the image.

6 References

[1]. J. A. Anderson, J. W. Silverstein, S. A. Ritz, and R. S. Jones, "Distinctive features, categorical perception, probability learning: Some applications of a neural model," in *Neurocomputing; Foundations of Research*, J. A. Anderson and E. Rosenfeld, Eds. Cambridge, MA: The MIT Press, 1989, ch. 22, pp. 283–325, reprint from *Psychological Review* 1977, vol. 84, pp. 413–451.

[2]. "*Associative Neural Memories: Theory and Implementation*," Mohamad H. Hassoun, Editor, Oxford University Press, 1993.

[3]. A. Schultz, "Collective recall via the Brain-State-in-a-Box network," *IEEE Transactions on Neural Networks*, vol. 4, no. 4, pp. 580–587, July 1993.

[4]. Thomas Chen, Ram Raghavan, Jason Dale, and Eiji Iwata. Cell Broadband Engine Architecture and its first implementation. Technical report, IBM, 2005. http://www.ibm.com/developerworks/power/library/pa-cellperf/

[5]. IBM. Cell Broadband Engine Architecture, http://www-128.ibm.com/developerworks/power/cell/docs documentation.html.
[6]. IBM. Cell Broadband Engine resource center. http://www-128.ibm.com/developerworks/power/cell/.
[7]. Annapolis Microsystems Inc, http://www.annapmicro.com/
[8]. Annapolis Wildstar-II pro data sheet, http://www.annapmicro.com/datasheets/ws2propci_markdata_13364_1_3.pdf
[9]. Xilinx Vertex-II pro devices, http://www.xilinx.com